Technical Report CS75021-R

A PROPOSAL FOR A MINILANGUAGE FOR THE BASIS OF DISCUSSION REGARDING DATA STRUCTURES

John A. N. Lee

Language Research Center
Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061

October 1975

The work reported herein was sponsored in part by the National Science Foundation Grant No. DCR74-18108.

Abstract

This report generalizes the concepts of data structures which were presented in mini-language 8 by Ledgard [1], and provides a basis for the more general discussion of data structures.

Keywords: Mini-languages, data structures, programming languages, directed graphs.

CR categories: 4.2, 4.29, 4.34

A PROPOSAL FOR A MINILANGUAGE FOR THE BASIS OF DISCUSSION REGARDING DATA STRUCTURES

INTRODUCTION

The discussion of elements of programming languages is too often stifled by the selection of current implemented languages such as ALGOL, FORTRAN, PL/I, etc., as the basis for exemplification. In these contexts the influences of prior experiences have been at work and have manifested themselves in the particular features that the languages possess. Thus any attempt to discuss language elements in terms of empirical implemented languages is doomed to eventually discuss our prior inabilities to forsee the potential uses of the languages. Furthermore, implemented languages are, by necessity, controlled in their scope by two facts of life; the current intellectual capabilities of the programmer and the current logical capabilities of the host equipment. As each of these capabilities is extended, so languages are similarly extended. In the same way, to discuss the domain of data structure language features in a limited context such as a language which admits only one type of structure is defeating the purpose of such discussions before the discussion commences. This paper proposes a "minilanguage" devoid of specific implementation schema and separate from other language features which operate over the products of data structure operations or which generate data which are input to data constructive operations.

The critical issues related to data structures are: *naming* and *description* of structures and their components, the *manipulation* of structures and their components including deletion of elements, the modification of elements and the insertion of elements, and the *sharing* of structures and their components by other structures. Typically, in the past, structures have been language restricted to be either regular arrays (including the COBOL or PL/I style of data structures) and linked lists (including various representations or implementations of rings). In most cases such structures have been implemented by the use of two elements; a (somewhat) ordered parent structure and a mapping function between the idealized, programmer's structure and that parent structure. Typically the parent structure has been some manifestation of a linearly addressed memory system. Thus the majority of our interest has been placed on the mapping function and its properties rather than on the structures and their benefits to the solution of pressing problems. This same bi-element system has lead to a discrimination between problems and studies of the individual problems rather than studies of the use of generalized data structures. In this paper a structure is proposed which is embedded in a skeletal language; the structures capable of being described, named, manipulated and shared in this language can cover an extremely wide range of classical structures and in fact can represent structures which, so far, have not yet appeared in production languages. Such structures include sets and content addressable systems.

STRUCTURES AS DIRECTED GRAPHS

The structure which is chosen as the basis for this elemental language is a single level directed graph composed of a root node, a set of distinctly named edges (distinct in the context of this one level graph), and a set of component nodes. Each single level graph is represented graphically as a tree, classically drawn upside down, from the root node of which emanate the named edges, at the termination of which are located the component nodes. Reference to the name of the structure implies the whole structure, being the root node, the edges and the component nodes. A component node is referenced by naming the edge on which it is located in the context of a named structure.

Component nodes may take two forms; they may be single level directed graphs themselves or they may be primitive elements in the universe of discourse. Such primitive elements are not structural and therefore cannot be considered to contain components or be capable of being divided into other elements. The universe of discourse can be one of two classes; in the context of operations over structures the universe is a class of values such as numbers, pointers or strings, whilst in the context of structure description the universe will be the class of structure descriptions including the primitive descriptors *number*, *pointer*, or *string*.

As a primary restriction we shall not permit structures to by cyclic; that is, a structure cannot be constructed which contains itself as a component (that is, as the component node at the termination of an edge emanating from its root node), or as the component of any of its components. This does not prevent a component which is a pointer from pointing to its containing structure or even to one of its own components; the restriction merely does not allow a structure to recurse on itself.

In the discussion which follows we shall restrict the domain of values to two primitive elemental types; number and pointer, represented in the language by *num* and *ptr* respectively.

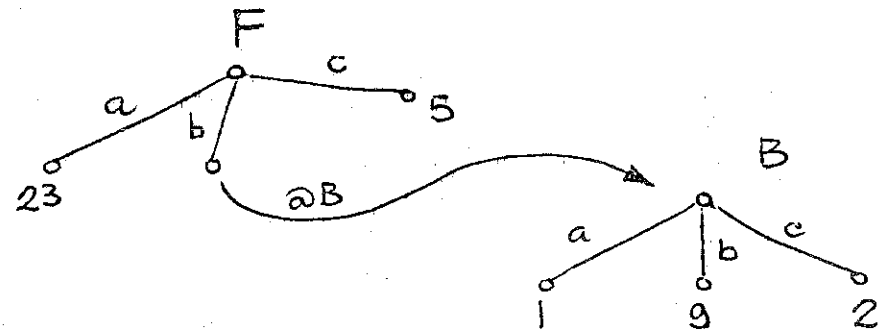


Figure 1a

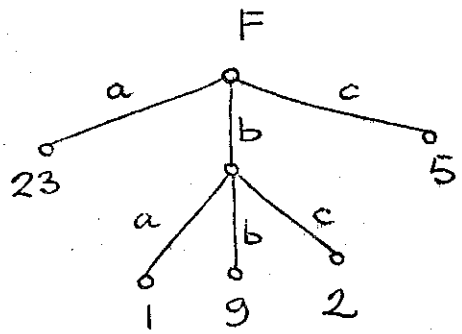


Figure 1b

THE LANGUAGE

To provide facilities related to the critical issues pertinant to data structures as mentioned earlier, the language will be composed of six basic elements:

- the description and naming of structure classes,
- the composition and naming of structures,
- the selection of structure components,
- a dynamic type statement and a type check to measure the conformance of a structure with a named description,
- a conditional statement, and
- a transfer of control system.

The generic, syntactic form of the representation of a structure is to be a set of edge name/component pairs; where in the case of a descriptor (*declaration* in programming language terms) the component is either a named descriptor or (in the case of a component which is value bound for the duration of the execution of the program) is a value descriptor. In the case of a manipulative expression (to be detailed later) the component in this pair will be a value, including another structure. Providing some syntactic sugaring to this basic form we generate an expression of the form

$$[c_1:d_1, c_2:d_2, \ldots, c_n:d_n]$$

where the c_i are the edge names (which we shall term *component identifiers*) which identify component structures and d_i are component or value descriptors depending on the context of the expression.

The basic declaration is given by the BNF syntax

<structure descriptor>::=dec <structure identifier>=
[<descriptor sequence>]

<descriptor>::= <component identifier> : <component descriptor>

<descriptor sequence> ::= <descriptor>{,<descriptor sequence>}$_0^1$

where <structure identifier> ::= {A|B|C|...|X|Y|Z}$_2^{>2}$

<component identifier> ::= {a|b|c|...|x|y|z}$_1^{>1}$

<component descriptor> ::= num | ptr | <structure identifier> |
<primitive expression>

The latter definition provides for a component descriptor which may take the form of an expression containing identifiers which yields on evaluation either a numeric value or a pointer to some structure. This provides for the case of a structure whose component is *always* (always being the duration of the program) directly related to properties of other structures, but whose component value is not *fixed*. This might seem to require the implementation of the language to continuously monitor the elements which would cause the modification of the value of such a component; in fact, any value evaluations need only be completed when there is a reference to the described structures. A simple tagging scheme to identify the components needing evaluation would resolve this problem. Where the primitive expression results in a pointer, the object of which is a structure which may or may not exist at any instant in time, the same strategy will suffice. However, we will specify that if the object of the pointer is currently non-existant, the value of the pointer shall be the null pointer.

To provide for alternative forms of descriptor, that is to permit a union of classes of descriptors, we shall provide a secondary declarative statement of the form

$$\underline{\text{dec}} \quad s = s_1 \vee s_2 \vee \ldots \vee s_n$$

where s and s_i are structure identifiers, those on the right hand side of the = sign being described in other descriptors elsewhere in the program.

Whilst a descriptor only specifies classes of structures and their associated names (*structure identifiers*), the basic operations of this language must be the construction and naming of structures. We choose to use the same syntactic form used for description of structures for specifying the structure itself, that is $[c_1{:}v_1, c_2{:}v_2, \ldots, c_n{:}v_n]$ where the c_i are component identifiers and the v_i are expressions which result in some structural or primitive object. Embedding this as the right hand side of an assignment statement, we provide both the mechanism for constructing structures and the means of naming those structures. The left hand side of an assignment statement may take one of two forms; either an (upper case single alphabetic character) identifier or a component expression. The former syntactic form represents the name which is to be associated with the structure created by the evaluation of the right hand part of the assignment statement, this action causing the removal of any association of that name with some other structure. It is not required that identifier names be at all times associated with some fixed structure descriptor; in fact, identifiers may be associated with structures for which there never exists an explicit structure descriptor.

A component expression is given the syntactic form

<component identifier> (<expression>)

where, in general, the expression enclosed inparentheses may be any expression which results in a structure. On the right hand side of an assignment statement this component expression will yield the named component of the structure; where either the structure does not contain an edge so named or the evaluation of the parenthesized expression does not result in a structure (that is the result is a primitive object), the application of the component identifier will yield the null object. On the left hand side of an assignment statement, the expression enclosed in parentheses should be limited to an identifier, or another component expression which satisfies the same criterion.

When the left hand portion of an assignment statement is a component expression the value of the right hand side then replaces that component in the named structure. Two other cases are of importance. If the component does not exist in the structure prior to the execution of the assignment statement, then that new component is added to the existing structure. For example, after the execution of the statement

F := [a:23, b:@B, c:5]

the structure shown in Figure 1a would be constructed. Subsequently, the execution of the statement

e(F) := @F

would modify this structure to that shown in Figure 2.

FIGURE 2

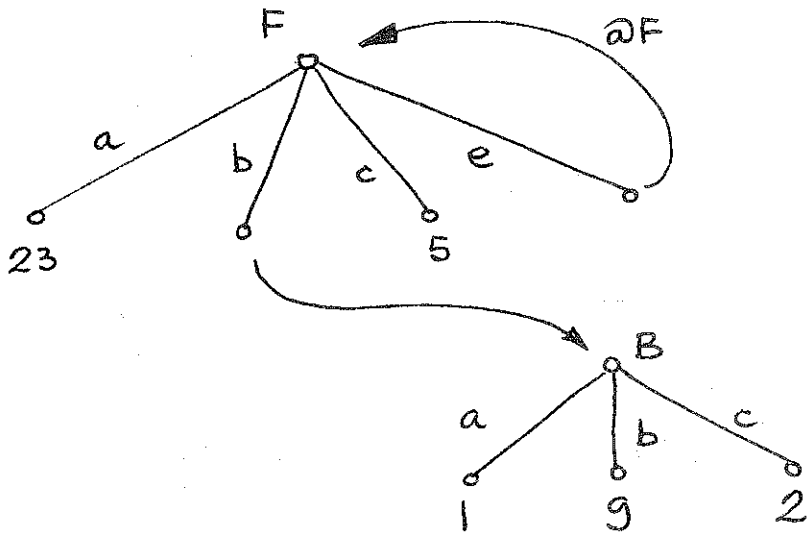
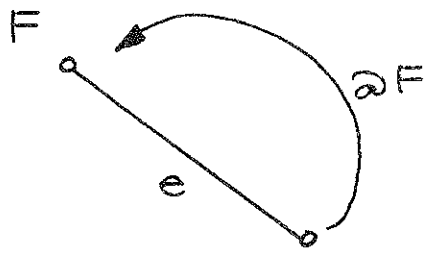


FIGURE 3



Similarly, if the structure named F did not exist previously to the execution of this statement, then the structure shown in Figure 3 would have been created.

Introducing the null object Ω, we may also provide for the removal or deletion of structures or their components. We prescribe that where the component is null (*not* the null pointer) the edge and its component may be considered to be removed from the structure.

The ability to make decisions within this language is based on the predicate which is formed from a structure identifier and the prefix is-. The application of a predicate to a structure results in a truth value which is then used to direct further program actions. The predicate expression has the syntactic form

is-<structure identifier> (<expression>)

The predicate expression is the first element in the conditional statement if <predicate expression> then <statement> else <statement> where <statement> may be any of the executable statement forms of the language.

The predicate expression also can stand as a statement in the language which expresses the applicable predicates over a structure. Semantically the stand alone predicate expression states that from this logical point forward, the predicate expression is true; if the case occurs where the predicate expression is no longer true then the program is not valid.

The subject of a predicate expression in this context is normally expected to be either the name of a structure or of a component of a structure. Subsequent to the statement, the named structure or component will always conform to the specified description. This implies that the implementation of the language will continually monitor the named objects to ensure their conformance. In practice such checks are necessary only when there is a reference to the named object either as a source of data (when the check should occur prior to the use of the data) or as a destination of data (when the check should occur subsequent to the store operation). There is not provided in the language a complementary predicate expression such as not-, nor is there provided a statement which removes the conformance requirements of a predicate expression. Instead, a simple renaming will remove the conformance requirements.

The final elements of this minilanguage are the statement label and the goto statement. For ease of reading we specify that each statement in the language (except declarations) may be preceded by an alphabetic label (upper case herein) and separated from the body of the statement by a colon. The goto statement then references the label and upon execution causes a transfer of control to the statement which is prefaced by the referenced label.

We have purposely omitted detailed reference to and description of the expressions which may occur in this language. Several forms are inherent in the language; expressions which construct structures, component expressions, simple identifiers, numeric value representations, and pointer expressions. Other forms of expression may be added to this list including (say) arithmetic expressions, but we shall not specify here such niceties which are truly outside the domain of this language.

Pointer expressions must result, on evaluation, in pointer values. However, where the expression contained in the parentheses associated with a pointer expression is a constructive expression, then the result is a pointer to an *unnamed* structure. That is, for example, the two sequential statements

```
X := [first:1,second:2]
Y := @X
```

are not equivalent to the single statement

```
Y := @ [first:1,second:2]
```

since in the second case the constructed structure is unnamed and therefore not referenceable directly elsewhere in the program. One more operator is necessary to overcome some semantic ambiguities related to pointer values. Let us assume that there exists a structure identified by A; then the statement

```
B := @A
```

assigns to the identifier B a pointer to the structure A. The question now arises as to the meaning of a reference to B. Should such a reference refer to A or the contents of B, that is, the pointer value? To overcome this ambiguity we shall specify that references to identifiers are meant to refer to the associated value (or structure). Thus following the execution of the above assignment statement, B refers to the value @A not to the structure A. Explicit references to the "pointed to" element by the use of a pointer value must use the operator val which strips off one level of indirection. Thus val(B) above would reference the structure named A. If B were not associated with a pointer value, the result of evaluating val(B) would be undefined.

References

[1] Ledgard, H. F., Ten Mini-Languages, Computing Surveys, V3 N2, Sept. 1971